Problem 11 Editorial: 'Tiger's *Tetris* Tournament Trickster
10 Points

Problem ID: `tetris`
Rank: `3`

Overview

Problem statement: https://calico.berkeley.edu/files/calico-fa22/contest/tetris/tetris.pdf

Although this problem was inspired independently, another problem writer soon identified the similar problem "Tetris Generation" (2021 ICPC Pacific Northwest Regionals Problem Q). The ICPC problem's mechanics are identical to those of our problem excepting how the ICPC problem allows the queue to begin mid-bag and excludes holds. However, these differences induce fundamentally different solutions.

The intended solution—and the only one we considered before the contest—combines greedy algorithms and ad hoc reasoning. Originally classified as Rank 2 under the assumption that implementation would be straightforward, this problem was promoted to Rank 3 after we spent hours debugging and checking our reasoning. We concluded that the ad hoc component proved unconventional and tricky enough to exceed any superficial greedy structure expected of USACO Bronze or Silver problems. Completely unexpectedly, among in-contest solutions, roughly half implemented a generic bitmask DP instead. There were also many others, which are difficult to cleanly categorize.

The author found inspiration for this problem very early in the problem writing process; it was the first ever proposal he submitted to CALICO! The communities of competitive programmers and *Tetris* players share a surprisingly wide overlap, plausibly from a common emphasis on pattern recognition. Lastly, we coincidentally had lots of relevant context to derive flavor text from: besides the associations between Qepsi, CodeTiger, LIT, and *TETR.IO*, `LIT` also happens to be a string of *Tetris* piece names. Our only concern was that Qepsi's ranking could drop during the contest, so we asked her directly to abstain from playing. This was our most well-received problem—the only receiving no dislikes in the feedback form—and we also loved writing it!

Main Test Set

Define B as the set of pieces in each bag. An implicit constraint in this problem is that there are $|B| = 7$ distinct pieces. The following discussion generalizes the problem to allow an arbitrary $|B| \geq 1$, a critical distinction when analyzing time and space complexity. See the end of the bitmask DP approach for a detailed discussion of complexity.

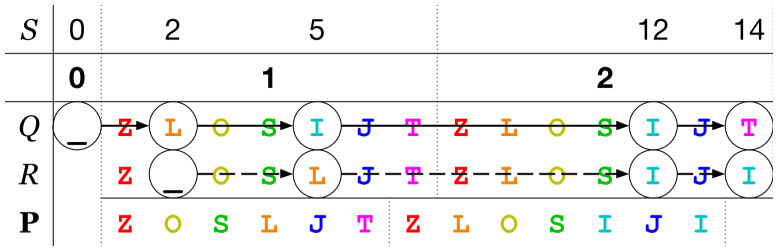
Greedy and Ad Hoc *Solutions available in C++, Java, Python*

Because *Tetris* is a video game, it has an engine that sequentially processes user inputs, motivating a solution based on simulating specific ordered bags and moves. We thus initialize a set $\varnothing \rightarrow U$, an initially empty set of the pieces used from the current bag. We also initialize a character '?' $\rightarrow h$, a hold slot initialized as a wildcard to be resolved later. In general, ? will mean that either the hold slot is empty or the hold slot contains an unknown piece, an equivalence that is more easily understood after considering all cases.

As an intuitive aid, consider the following visualization of a simulation:

- Q is defined as the finite prefix of the queue with all pieces that are ever held or placed before all pieces in $\mathbf{P}$ have been placed. It contains zero or more full bags, possibly followed by a partial bag once no more pieces are needed; bags are indexed "bag 1", "bag 2", …, and pieces are indexed q_1, …, q_k. To allow for holds, Q is preceded by an imaginary "bag 0" we define as containing only the placeholder q_0 = '_', encoding an empty hold slot.
- $S = (s_0, ..., s_l)$ is defined as a strictly increasing sequence of hold indices: Holds are applied to the pieces in Q indexed s_0, ..., s_l. The hold slot starts empty, which can be denoted with the convention that $s_0 = 0$. Each hold can then be drawn as an arrow pointing from an original held piece q_{s_i} to a new held piece $q_{s_{i+1}}$.
- We define one more sequence $R = (r_1, ..., r_k)$, absent from the statement. If a piece is placed directly, the piece is copied to R. If a hold was used, the original held piece is copied to R. This mapping is a bijection between Q (excluding bag 0) and R.

Any valid playthrough is fully parameterized by a unique ordered pair (Q, S), inducing some R and $\mathbf{P}$. The ordered pair ('_ZLOSIJTZLOSIJT', (0, 2, 5, 12, 14)) is visualized as follows:



Our visualization illustrates many observations about the nature of holding:

1. Arrows are drawn head-to-tail, forming a "chain" of held pieces over Q. The chain is then shifted rightwards by one "link" to create the dashed chain over R.
2. Any _ behaves like a "gap" in R. $\mathbf{P}$ is the "compression" of R induced by deleting the _ from R if present.
   a. If two sequences R and R' deviate only by the presence or position of the _, they induce the same $\mathbf{P}$.
   b. The first hold disrupts the "synchronization" between R and $\mathbf{P}$, seemingly shifting $\mathbf{P}$ leftwards by one space to fill the gap such that $\mathbf{P}$ is always one piece "ahead" of Q. Subsequent holds preserve the existing synchronization.
3. If the current and held pieces match before a hold, the hold arrow links two identical pieces, so the game state is unchanged after holding.

For each given piece $\mathbf{P}_i$, we must decide the piece's source, either the current bag (by placing directly) or the hold slot (by holding before placing). Observation 3 hints that the source can be selected by a greedy algorithm; if, through simplifying observations, we can decide what the next move must be without needing to consider far into the past or future queue, the final time complexity will be linear in $\mathbf{N}$.

Processing each $\mathbf{P}_i$ sequentially, all scenarios can be categorized under one of four cases. Implement the casework as an `if/else` chain so that only the first matching case is applied.

1. If $\mathbf{P}_i \notin U$, $\mathbf{P}_i$ can come from the current bag. We will use casework to show that it is always optimal here to take from the bag, so insert $\mathbf{P}_i$ in U. In general, we will show that greedily minimizing the number of holds is a valid strategy.
   a. If the hold slot is empty, $\mathbf{P}_i$ cannot come from it.
      i. If $|U| < |B| - 1$, multiple pieces remain in the bag. The next queued piece after $\mathbf{P}_i$ thus differs from $\mathbf{P}_i$, so holding would be invalid.
      ii. If $|U| = |B| - 1$, $\mathbf{P}_i$ is the last piece in the bag. The player could hold this last piece and wait until the first piece of the next bag to place $\mathbf{P}_i$, but we claim this is unnecessary.
      Proof. Assume (Q, S) induces R and $\mathbf{P}$, and the first hold was used on the last piece of bag j (formally, $s_1 = |B|j$.) The "gap" (from Observation 2) must now be compensated for by immediately placing the queue piece from index s_1 in one of two ways: If a second hold was used on index $s_2 = s_1 + 1$, the held piece is q_{s_1+1} after using that piece (*). If no such hold was used, we must have $q_{s_1+1} = q_{s_1}$, so the held piece is also q_{s_1+1} (**). From Observation 2a, we can now construct an

alternative parameterization (Q, S') inducing R' and $\mathbf{P}$: let $S' := (s_0, s_2, ..., s_l)$ if (*) or $S' := (s_0, s_1 + 1, s_2, ..., s_l)$ if (**). In both cases, R and R' differ only in that the _ occurs one index later in R'. The hold sequence and pieces are synchronized between both parameterizations after index $s_1 + 1$ of Q, so we can copy the rest of the original parameterization. ∎

   b. If the hold slot is full but does not hold $\mathbf{P}_i$, holding immediately invalidates the simulation.
   c. If the current and held pieces are both $\mathbf{P}_i$, by Observation 3, we can still take from the bag.
2. If $\mathbf{P}_i = h$ (implicitly requiring $h \neq$ '?'), $\mathbf{P}_i$ can only come from the hold slot. Although it may seem like we should immediately change h to the piece being swapped in (the current piece), we generally do not know the current piece: If there are multiple pieces left in the bag, the current piece could be any of them. For now, assign '?' $\to h$, here representing a placeholder for a hold slot known to be full.
3. If $h =$ '?' and $|U| = |B| - 1$, we can now determine h: only one piece c in the current bag has not been used, so it must be the uncounted hold piece if the hold slot is full, since the unknown piece was held from the current bag. Assign $c \to h$. Now that c has been used, the bag would be refilled ($\varnothing \to U$), except that $\mathbf{P}_i$ still needs to be taken to finish this iteration, and since this situation fits Case 1's criteria, also assign $\{\mathbf{P}_i\} \to U$.
4. If none of the previous cases match, we claim that the sequence is impossible.
   Proof. We now know that $\mathbf{P}_i$ does not remain in the bag and is not a known held piece, leaving the case where we assigned $h =$ '?' somewhere in the bag. Given that we are sure $\mathbf{P}_i \in U$, the desired piece has been placed earlier and is no longer available. ∎

At the end of each iteration, if $|U| = |B|$, the bag has been depleted, and we must clear U.

We now explain why $h =$ '?' can also account for an empty hold slot. In both contexts for ?, the hold slot does not contain a piece that we guarantee can be used immediately, meaning it can only help us continue the construction of $\mathbf{P}$ after we can determine its value in Case 3. This is a deep observation; one problem writer initially implemented this solution starting with $h =$ '_' and adding additional `if`/`else` clauses. A second problem writer accidentally implemented the solution without that extra case, but both writers eventually realized that the second solution was actually correct.

Time: $O(\mathbf{N} \log |B|)$ *Memory:* $O(\mathbf{N} + |B|)$

Bitmask DP *Solution available in C++ (in-contest solution by CodeTiger)*

Special thanks to CodeTiger (Alex Fan) of team OlyFans for providing some extra insight and allowing us to use his code. We completely preserved the formatting to demonstrate how this approach could be used in-contest.

Approaching through bitmask DP, this problem becomes mostly implementation, albeit a much more involved one than the greedy solution. From a contestant's perspective, trying to find an elegant implementation is often risky: Such a search can take indefinitely long, while implementing a known algorithm is guaranteed to succeed within a fixed (possibly long) time. This risk, contextualized within the pressure of a live contest, explains the popularity of this approach.

Begin by initializing a DP array with three dimensions.

Dimension #	Index Count	Meaning of Each Index
1	$\mathbf{N}$	A piece index i
2	$2^{\lvert B\rvert}$	A subset of B indexed as a bitmask
3	$\lvert B\rvert + 1$	A value of the hold slot (one of the $\lvert B\rvert$ pieces or empty)

Computing the final array thus takes $O(\mathbf{N}\lvert B\rvert 2^{\lvert B\rvert})$ time and memory. Because $\lvert B\rvert = 7$ in standard *Tetris*, the execution time for this seemingly exponential-time algorithm (≈0.2 s) is still far below the time limit, albeit about 20 times higher than from the greedy approach (≈0.01 s). Most programs submitted with this approach also finished under the memory limit, yet by a thin margin. At least one team's program seemed to work on their machines but received a “Runtime Error” verdict as a result of exceeding the memory limit during judging; only when that team optimized their code by a constant factor after the contest did they receive an “Accepted” verdict.

Had we predicted this approach during problem writing, we might have added a bonus test set where the 18 pentominoes are used instead of the 7 tetrominoes. Then, the intended greedy solution would still pass comfortably, while the bitmask DP solution would far exceed both time and memory limits.

Time: $O(\mathbf{N}\lvert B\rvert 2^{\lvert B\rvert})$ *Memory:* $O(\mathbf{N}\lvert B\rvert 2^{\lvert B\rvert})$